On Temporal Verification of Stateful P4 Programs

Delong Zhang, Chong Ye, and Fei He,
School of Software, BNRist, Tsinghua University, Beijing 100084, China;
Key Laboratory for Information System Security, MoE, China

https://www.usenix.org/conference/nsdi25/presentation/zhang-delong

This paper is included in the
Proceedings of the 22nd USENIX Symposium on
Networked Systems Design and Implementation.

April 28–30, 2025 • Philadelphia, PA, USA

978-1-939133-46-5

Open access to the Proceedings of the
22nd USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

On Temporal Verification of Stateful P4 Programs

Delong Zhang* Chong Ye* Fei He†
School of Software, BNRist, Tsinghua University, Beijing 100084, China
Key Laboratory for Information System Security, MoE, China

Abstract

Stateful P4 programs offload network states from the control plane to the data plane, enabling unprecedented network programmability. However, existing P4 verifiers overapproximate the stateful nature of P4 programs and are inherently inadequate for verifying network functions that require stateful decision-making.

To overcome this limitation, this paper introduces an innovative approach to verify P4 programs while accounting for their stateful feature. We propose a specification language named P4LTL, tailored for describing temporal properties of stateful P4 programs at the packet processing level. Additionally, we introduce a novel concept called the *Büchi transaction*, representing the product of the P4 program and the P4LTL specification. The P4 program verification problem can be reduced to determining the existence of any fair and feasible trace within the Büchi transaction. To the best of our knowledge, our approach represents the first endeavor in temporal verification of stateful P4 programs at the packet processing level. We implemented a prototype tool called p4tv. Evaluation results demonstrate p4tv's effectiveness and efficiency in temporal verification of stateful P4 programs.

1 Introduction

Software Defined Networking (SDN) [22] is an innovative network architecture that offers unprecedented network flexibility and programmability by decoupling the control plane and data plane. In traditional SDN architecture, the data plane's sole responsibility is forwarding packets [30]. Unfortunately, this configuration often results in frequent communication between the control and data planes, leading to significant network delay and overhead [55].

Recently, a more intelligent data plane has emerged, capable of maintaining network states locally to determine forwarding behaviors [9, 32, 38, 43, 45, 46, 49]. This novel architecture is known as the *stateful data plane* [55]. By updating the states of the data plane locally and adjusting forwarding behaviors based on these local states, the architecture reduces the excessive controller interaction. As a result, it offers faster response time and enhances network flexibility. In this paper, we focus on P4 [10], a widely adopted data plane language that supports the stateful data plane.

Formal methods play a pivotal role in ensuring the correctness of P4 programs. In recent years, numerous P4 program verifiers have been developed, including ASSERT-P4 [24], Aquila [50], bf4 [21], p4v [40], P4-NOD [41], and Vera [47]. However, these existing P4 program verifiers have primarily focused on per-packet processing. They treat the register states of P4 programs as fully nondeterministic at the beginning of each packet processing, simplifying the checks for continuous packet processing events into a single check. Consequently, they overlook the preservation of network states, treating each packet processing in isolation. As a result, these P4 program verifiers are inherently inadequate for verifying network functions that require stateful decision-making [31]. For example, a simple functionality like "if link failure is detected, then packets arriving *afterward* will all be forwarded to the backup port" [38, 45, 49] is not currently expressible and verifiable by existing verifiers.

Verifying P4 programs while accounting for their stateful nature presents significant challenges. The first challenge relates to environment modeling. It is important to note that P4 programs are deployed within switches, which continuously receive packets from the network and invoke the P4 programs to process these packets. The processing of packets by the P4 program can be influenced by the network states, and the behaviors of the P4 program for different packets can be interrelated. Therefore, when verifying the correctness of a P4 program, it is inadequate to consider its behaviors solely in the context of processing a single packet (as in the existing approach [24, 40, 47, 50]); instead, we should consider its behaviors in processing a series of packets. To address this challenge, we propose the use of an environment model (in Section 3) and verify the behaviors of P4 programs within the

*Both authors contribute equally to this work.
†Fei He is the corresponding author.

context of this environment model.

The second challenge pertains to the specification language used to describe the behaviors of a P4 program as it processes a series of packets. Unlike current P4 verifiers [21, 24, 40], which primarily focus on verifying properties for individual packet processing, our task now requires specifying properties that span multiple packet processings. This is considerably more challenging as it demands consideration of multiple executions of the P4 program. Existing data plane specification languages tend to either be too high-level, focusing on events of the entire network but not the executions of the P4 programs (e.g., NetKAT [3], SNAP [4], and NICE [11]), or too low-level, focusing solely on events within individual packet processing (e.g., NetCTL [47]). Although some simple safety properties (i.e., ensuring that something bad never happens [35]) can be reformulated into assertions for a single packet processing, the specification language for individual packet processing generally lacks the capacity to depict complex properties, especially those related to liveness (i.e., ensuring that something good eventually happens [35]). To tackle this challenge, we extend Linear Temporal Logic (LTL) [44] and introduce a specification language called P4LTL (in Section 4) that facilitates specifying temporal properties of P4 programs at the packet processing level.

Throughout the paper, we use the term *temporal verification of stateful P4 programs* to refer to the verification of temporal properties that span multiple packet processings. The third challenge lies in the development of algorithms for conducting such verification. It is important to note that the granularity of time steps plays a crucial role in temporal verification. In our specific context, we need to consider two distinct time steps: the time step that occurs after the execution of a P4 program statement, referred to as the *small time step*, and the time step that occurs after processing a packet, referred to as the *big time step*. Notably, a big time step comprises multiple small time steps. We introduce a novel concept known as a *Büchi transaction*, representing the product of the P4 program and the P4LTL specification. This concept is carefully defined to harmonize these two distinct time steps. We prove that the P4 program satisfies the P4LTL specification if and only if the Büchi transaction does not exhibit any fair and feasible trace. Building upon this observation, we develop an efficient verification algorithm (in Section 5) for temporal verification of stateful P4 programs.

With the above techniques, we develop p4tv, a temporal verifier tailored for stateful P4 programs. The framework of p4tv is illustrated in Figure 1. Given a P4 program and a P4 specification as inputs, it either returns "Verified" if the program meets the specified property or provides a counterexample that exposes a defect in the program.

We manually collected 9 benchmarks of stateful P4 programs from open-source communities and formulated 19 temporal verification tasks to evaluate p4tv. Experiment results demonstrate both effectiveness and efficiency of p4tv in verifying the temporal properties of stateful P4 programs. Furthermore, we conducted experiments to evaluate p4tv's efficiency in assertion verification and to assess its scalability concerning program complexity and specification complexity.

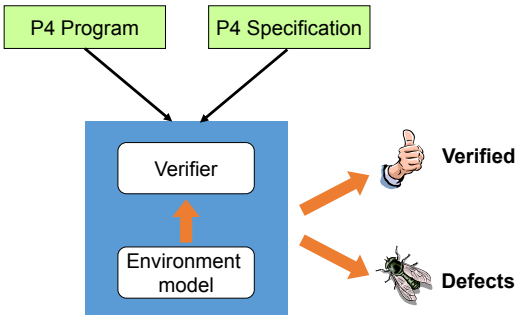

Figure 1: The workflow of p4tv .

Our contributions can be summarized as follows:

- We present, to the best of our knowledge, the first approach for temporal verification of stateful P4 programs.
- We design P4LTL, a specification language for describing temporal properties of stateful P4 programs at the level of packet processing.
- We propose a novel concept called the Büchi transaction, representing the product of a P4 program and a P4 specification. This concept allows us to reduce the P4 program verification problem to determining the existence of any fair and feasible trace within the Büchi transaction.
- We implement a prototype tool called p4tv and show its effectiveness on publicly-accessible benchmarks.

2 Motivations

In this section, we present the processors-based network immune scheme (P4NIS) mechanism [38] to provide motivation for our approach. P4NIS is designed to safeguard the network from packet eavesdropping within the programmable data plane. A snippet of P4NIS code (slightly modified for clarity) is depicted in Figure 2, where *count* serves as a register variable and *check_passed* validates whether all forwarding conditions are met. This validation includes a packet validity check (i.e., the ingress port and header validity check) and a range check on *count* (i.e., $0 \leq count \leq 2$). Upon the arrival of a packet that passes *check_passed*, the switch circularly distributes the packet traffic and executes various eavesdropping defense mechanisms.

One essential safety property for P4NIS states: *"The forwarding ports for valid incoming packets should always remain within the valid range (1 to 3 inclusively)."*

Nevertheless, existing P4 verifiers assess P4 programs on a single-packet basis, abstracting registers as fully nondeterministic. Consequently, these verifiers will report that the property

```
1  count.read(temp,(bit<32>)0);
2  if(check_passed()){
3      // send packages circularly
4      temp = temp + 1;
5      standard_metadata.egress_spec = temp;
6      if (temp == 3) temp = 0;
7      count.write((bit<32>)0,(bit<32>) temp);
8      // other operations
9      ...
10 }
```

Figure 2: The snippet of P4NIS code.

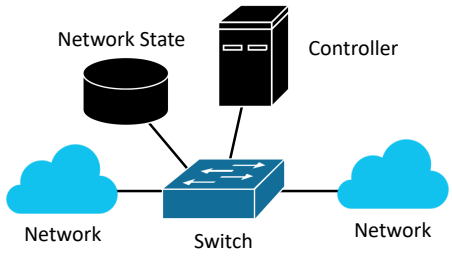

Figure 3: The environment model of p4tv.

does not hold for P4NIS. This is evidenced by a counterexample where the register *count* is initially set to a value of 4, bypassing the condition check (at line 2) and resulting in an implicit drop.

However, to reason about the precise behaviors of stateful P4 programs, it is necessary to analyze them across multiple packets. Considering the P4NIS example, if the register *count* is initialized with 0 during network setup, as per the code from lines 4 to 7, the value of *count* cannot exceed the valid range regardless of the number of packets processed by this P4 program, thereby proving the property. This underscores the importance of assessing P4 programs in the context of multiple packets.

One could suggest revising the property to: *"If the value of count is valid at the start of processing, the forwarding ports will remain valid."* While this simplifies verification into a single-packet check, identifying the specific precondition (i.e., the value of *count* falls within the valid range) requires substantial expertise and manual effort.

Furthermore, there exist properties that can hardly be simplified into a single-packet check. For instance, consider the liveness property for P4NIS that ensures the consistent activation of all defense mechanisms: *"When valid packets arrive infinitely often, they should be forwarded to every valid port infinitely often."* Expressing liveness concept like "infinitely often" in the context of a single-packet check is challenging.

These challenges motivate us to model the P4 execution process in a stateful manner (Section 3), introduce specification language P4LTL for stateful P4 programs (Section 4), and propose its corresponding automated verification algorithm (Section 5). Leveraging these techniques, p4tv is able to effectively and automatically verify complex temporal properties.

3 Environment Model

A P4 program specifies the behavior of a switch upon receiving a packet. To ensure the correctness of P4 programs, it is necessary to specify their environment model. As illustrated in Figure 3, the programmed switch receives packets from the network and processes them based on rules provided by the controller and local network states. In Section 3.1, we present our modeling mechanism for network environment and register states in comparison with existing P4 verifiers. In Section 3.2, we introduce our control plane interface which models the controller part.

3.1 The Basic Model

The modeling mechanism employed in current P4 verifiers can be seen as follows:

```
1  procedure main(){
2      nondet_register_variables();
3      call P4Program();
4  }
```

The model only considers the processing of a single packet and assumes that register variables are fully nondeterministic. However, this level of abstraction overlooks the stateful nature of packet processing, where register values in fact persist and are preserved for subsequent processing.

The real-world switch may receive arbitrarily many packets from the network environment. As a result, the P4 program needs to be executed arbitrarily many times. To model the stateful nature of packet processing, we regard a P4 program as an infinite while loop, similar to the idea presented in the probabilistic P4 test tool [31]. Since we consider the correctness of the P4 program, the network environment should not be constrained and the input packets should be nondeterministic. Therefore, we add an initialization block at the entry of the while-loop, in which we assign each packet field a nondeterministic value. The metadata fields in P4 programs are initialized in accordance with the P4-16 specification [16]. Considering that the control plane may change constantly, the table applications are modeled with a nondeterministic action choice, similar to the technique used in p4v [40] and bf4 [21]. This approach ensures that the control plane remains unconstrained. Notably, we declare the registers outside the loop to keep their value after each iteration (i.e., each packet processing). Our modeling mechanism is as follows:

```
1  procedure main(){
2      declare_registers();
3      while(true){
4        packet_initialization();
5        call P4Program();
6      }
7  }
```

$$
\begin{array}{lcl}
\text{P4LTL}\,\varphi & \rightarrow & \psi \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \\
& & \mid \circ\varphi \mid \Diamond\varphi \mid \Box\varphi \mid \varphi\, \mathcal{U}\, \varphi \mid \varphi\, \mathcal{W}\, \varphi \\
\text{Pred}\,\psi & \rightarrow & t\,\text{comp}\,t \mid \text{fwd}(t) \mid \text{valid}(h) \mid \text{hit}(t) \mid \text{drop} \\
& & \mid \text{apply}(t,a) \mid \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \psi \rightarrow \psi \\
\text{Term}\,t & \rightarrow & d \mid h \mid m \mid v \mid r[t] \mid \text{old}(t) \mid \text{key}(t,k) \\
& & \mid \text{para}(t,a,p) \mid t\ \text{op}\ t
\end{array}
$$

Figure 4: Syntax of P4LTL.

It is important to note that our modeling mechanism differs from previous P4 verification studies. Existing P4 verifiers only consider a single packet processing, assuming register states are fully nondeterministic at the start of each packet processing. While this technique simplifies the verification process, it hinders the ability to track changes of register states throughout packet processings. On the other hand, we consider the execution sequences in a continuous and correlated manner, reflecting their actual occurrence. This approach models the stateful nature of packet processing, which is crucial for stateful P4 program analysis and verification.

3.2 Control Plane Interface

Some P4 functionalities cannot be properly verified without additional information regarding the control plane. Therefore, we present *control plane interface* (CPI). which facilitates users specifying the control plane assumption.

For instance, consider a scenario where a rule exists in the table *rt* that triggers the action fwd with the parameter *pt* set to 1 when the key $m.d$ equals 2. In this case, the corresponding rule can be expressed in CPI as follows:

$$
\Box(\text{hit}(rt) \wedge \text{key}(rt, m.d) == 2 \rightarrow \\
\text{apply}(rt, fwd) \wedge \text{para}(fwd, pt) == 1).
$$

CPI provides a novel feature that assists developers by allowing not only concrete table entries but also symbolic ones. For instance, developers can modify the equality condition above to inequality, creating a symbolic CPI. This symbolic CPI encompasses the set of rules triggering fwd when the key $m.d$ is not equal to 2. This allows developers to verify P4LTL under control plane snapshot (i.e., table rules) sets instead of a single control plane snapshot. A blank CPI simply indicates that every table rule is possible.

4 Specification Language

To specify temporal properties spanning states of packet processing in stateful P4 programs, we introduce P4LTL, a language that extends the well-established formalism of Linear Temporal Logic (LTL) [44].

Among various temporal logics such as CTL, LTL, and CTL* [51], all capable of expressing temporal properties, we choose LTL as the foundational logic of P4LTL for the following reasons: (1) its linear time framework aligns more naturally with our problem domain, and (2) its rich expressiveness allows for specifying intricate properties along the packet processing traces.

4.1 P4LTL

The syntax of P4LTL is shown in Figure 4. In this section, we describe the P4LTL design in a bottom-up fashion.

Terms. Terms of P4LTL mainly consist of integer numbers d, packet headers h, metadata m, header field h, free variables v and register r. They represent the corresponding values *at the end of* packet processing. Furthermore, $key(t,k)$ denotes the value of key k when applying table t, while $para(t,a,p)$ represents the value of parameter p for action a when invoked by table t. Additionally, P4LTL provides a special construct $old(t)$ to refer to the value *at the start of* the ingress pipeline (i.e., at the end of the parsing pipeline). For instance, we can use $old(r[i])$ and $old(h)$ to refer to the register value of r at index i and the header value at the start of core packet processing. [1] Moreover, P4LTL allows the use of arithmetic operations between two terms with the arithmetic operator op that is supported by P4. Notably, P4LTL supports the use of free variables v, whose values are non-deterministic and can be further constrained by predicates.

Predicates. P4LTL provides several predicates for users to describe packet forwarding behaviors. To be specific, the semantics of predicates are defined as follows:

1. Predicate $t\ comp\ t$ is true if the corresponding terms meet the situation in the packet forwarding process, where *comp* denotes the comparison operators in P4.
2. Predicate $valid(h)$ is true if the corresponding header h is valid after the parsing progress.
3. Predicate *drop* is true if the packet is dropped at the end of packet processing.
4. Predicate $fwd(t)$ is true if the packet is forwarded to port t at the end of packet processing.
5. Predicate $hit(t)$ is true if the table t is hit during the packet processing.
6. Predicate $apply(t,a)$ is true if the action a is invoked by table t during the packet processing.

Note that $fwd(t)$ can be rewritten as a special case of the predicate $t\,\text{comp}\,t$. In P4 programs, the forwarding port can be

[1] For the sake of completeness, we consider the values of $old(key(t,k))$ and $old(para(t,a,p))$ to be arbitrary.

represented as *standard_metadata.egress_spec*, so the predicate *standard_metadata.egress_spec* $== t$ is equivalent to $fwd(t)$. We present it as a syntactic sugar for convenience.

Time step for stateful P4 programs. The LTL component is the key part of P4LTL. As the semantics of LTL hinge on the idea of *execution trace*, it is necessary to define what its *time step* is at the outset. An execution trace constitutes a sequence of valuations, which map P4LTL terms to their values at the end of every packet processing. A time step is performed every time the P4 program finishes processing a packet. It is important to note that the semantics of P4LTL are different from LTL in standard program verification [17, 18, 20] or NetCTL [47]. P4LTL concentrates on temporal properties across multiple packet processing, whereas other approaches primarily center around temporal properties across statements or pipelines in a single packet processing.

P4LTL. Based on the illustrations of P4LTL time steps, we define the semantics of P4LTL operators.

- Operator *next* $\circ$ is satisfied when the condition holds at the end of the next packet processing.
- Operator *eventually* $\Diamond$ is satisfied when the condition holds at the end of some packet processing eventually.
- Operator *always* $\Box$ is satisfied when the condition holds at the end of every packet processing.
- Operator *until* $\mathcal{U}$ is satisfied if the left-hand condition stays true until the right-hand condition becomes true, and the right-hand condition eventually holds true.

We further introduce syntactic sugar *weak-until* $\mathcal{W}$ in the following manner: $\varphi_1 \mathcal{W} \varphi_2$ is equivalent to $(\varphi_1 \mathcal{U} \varphi_2) \vee \Box \varphi_1$. A trace τ satisfies a P4LTL specification φ is written as $\tau \models \varphi$. For a P4 program $\mathcal{P}$ and a P4LTL specification φ, $\mathcal{P} \models \varphi$ if and only if all $\mathcal{P}$'s feasible execution traces $\tau \models \varphi$.

CPI. To specify the control plane assumption, developers can leverage terms $key(t,k)$, $para(t,a,p)$ and predicates $hit(t)$, $apply(t,a)$, as exemplified in Section 3.2. As per the semantics of P4LTL, alterations in the control plane amidst a packet sequence do not affect the validity of the verified P4LTL property, as long as the control plane still complies with the symbolic CPI. We integrate the control plane assumption ψ_C along with the property φ_{old} to form the new P4LTL formula $\varphi_{new} = \psi_C \rightarrow \varphi_{old}$ and proceed with the verification.

Example. Consider the liveness property, "Specific packets arriving infinitely often should be forwarded to every valid port infinitely often," introduced previously in Section 2. We can express the property with P4LTL as follows. The environment assumption $\varphi_E = \Box\Diamond(\text{valid}(hdr.label))$ states that we only consider the situation when specific packets come infinitely often. The temporal property $\varphi_P = \Box(\neg \text{valid}(hdr.label) \rightarrow \text{drop}) \wedge \Box\Diamond \text{fwd}(1) \wedge \Box\Diamond \text{fwd}(2) \wedge \Box\Diamond \text{fwd}(3)$ specifies that the specific packets should be forwarded infinitely often to valid ports, while other packets are dropped. In the end, the P4LTL formula $\varphi = \varphi_E \rightarrow \varphi_P$ depicts the desired property.

5 Verification

In this section, we present our approach for temporal verification of the stateful P4 programs. A fundamental challenge arises from the fact that the semantics of P4 programs and the semantics of the P4LTL specification attribute different meanings to the concept of a time step. Consequently, there arises a necessity to align these two distinct time steps within our verification algorithm. To address the issue, we propose a novel concept Büchi transaction, extending the classical program LTL verification approach [20].

5.1 Büchi Transaction

Our approach is rooted in the verification paradigm proposed in [20], which relies on its definition of the Büchi program product.

The concept Büchi program product extends from the concept of *Büchi automaton*, designed to accept infinite words. A Büchi automaton is a tuple $\mathcal{A} = (\Sigma, Q, q_0, \delta, \mathcal{F})$, which consists of a finite alphabet Σ, a finite state sets Q with an initial state $q_0 \in Q$, a transition relation $\delta : Q \times \Sigma \rightarrow Q$ and a set of final states $\mathcal{F} \subseteq Q$. Büchi automaton $\mathcal{A}$ accepts a word $w \in \Sigma^\omega$ if its corresponding run on $\mathcal{A}$ contains any final states infinitely often. The word accepted by Büchi automaton can also be called a *fair* trace. In addition, each LTL formula can be transformed into an equivalent Büchi automaton following the well-established algorithm [25].

Definition. (Büchi program product) A program is a tuple $\mathcal{P} = (\mathcal{S}, \mathcal{L}, \ell_0, \delta_p)$ consisting of a statement set $\mathcal{S}$, a program location set $\mathcal{L}$ with program entry ℓ_0 and a transition relation $\delta_p : \mathcal{L} \times \mathcal{S} \rightarrow \mathcal{L}$. Given a specification Büchi automaton $\mathcal{A}_\varphi = (\Sigma, Q, q_0, \delta, \mathcal{F})$, the Büchi program product $\mathcal{P} \times \mathcal{A}_\varphi$ is a Büchi automaton $\mathcal{B} = \left(\widehat{\Sigma}, \widehat{Q}, \widehat{q_0}, \widehat{\delta}, \widehat{\mathcal{F}}\right)$ where:

- $\widehat{\Sigma} = \{s; \text{assume } (\phi) \mid s \in \mathcal{S} \wedge \phi \in \Sigma\}$
- $\widehat{Q} = \{(l,q) \mid l \in \mathcal{L} \wedge q \in Q\}$ and $\widehat{q_0} = (\ell_0, q_0)$
- $\widehat{\delta} = \{((l,q), s; \text{assume}(\phi), (l',q')) \mid (l,s,l') \in \delta_p \wedge (q,\phi,q') \in \delta\}$
- $\widehat{\mathcal{F}} = \{(\ell,q) \mid l \in \mathcal{L} \wedge q \in \mathcal{F}\}$

The definition of Büchi program product is actually the product of two automatons, the program $\mathcal{P}$ and the specification $\mathcal{A}_\varphi$. The product is also a Büchi automaton, which enables acceptance of infinite traces. Informally, one can view the Büchi program product of $\mathcal{P}$ and the specification negation $\mathcal{A}_{\neg\varphi}$ as the Büchi automaton that witnesses the violation, i.e., accepts counterexamples. It has been proven that every

fair and feasible trace of this Büchi program product is a counterexample demonstrating that $\mathcal{P}$ violates the specification φ in [20], where a corresponding checking algorithm is also proposed. A trace is *feasible* if the corresponding sequence of statements is executable. The notion executable means that the *assume* conditions along the execution are all satisfiable.

Yet by definition, Büchi program product only concerns traces at the *small time step* (i.e., at the level of statements), which can not be directly applied to verify P4LTL that concerns the *big time step* (i.e., at the level of packet processing). For brevity, we denote the execution of the P4 program in our execution model as a *transaction*. In P4LTL verification, we can not allow the LTL automaton to transition every time the control flow automaton transitions. The LTL automaton should only take its step to transition whenever the *transaction* of the control flow automaton finishes, in correspondence with the semantics of P4LTL. Thus, we extend Büchi automaton to *Büchi transaction*, aiming to check LTL properties at the packet processing level. Its formal definition is as follows:

Definition (Büchi Transaction) Consider a program $\mathcal{P} = (\mathcal{S}, \mathcal{L}, \ell_0, \delta_p)$. We call it $\mathrm{TE}(s)$/ $\mathrm{TR}(s)$ when $s \in \mathcal{S}$ is an entering/return statement to/from the transaction. Given a specification automaton $\mathcal{A}_\varphi = (\Sigma, Q, q_0, \delta, \mathcal{F})$, the Büchi transaction $\mathcal{P} \otimes \mathcal{A}_\varphi$ is a Büchi automaton $\mathcal{B} = \left(\widehat{\Sigma}, \widehat{Q}, \widehat{q_0}, \widehat{\delta}, \widehat{\mathcal{F}}\right)$ where:

- $\widehat{\Sigma} = \{s; \text{assume } (\phi) \mid s \in \mathcal{S} \wedge \phi \in \Sigma\}$
- $\widehat{Q} = \{(l,q) \mid l \in \mathcal{L} \wedge q \in Q\}$ and $\widehat{q_0} = (\ell_0, q_0)$
- $\widehat{\delta} = \{((l,q), s; \text{assume}\,(\phi), (l',q')) \mid$
  $(\mathrm{TR}(s) \wedge (l,s,l') \in \delta_p \wedge (q,\phi,q') \in \delta) \vee$
  $(\neg \mathrm{TR}(s) \wedge (l,s,l') \in \delta_p \wedge q = q' \wedge \phi = true)\}$
- $\widehat{\mathcal{F}} = \{(\ell,q) \mid q \in \mathcal{F} \wedge (\mathrm{TE}(\ell) \vee \mathrm{TR}(\ell))\}$, where
  - $\mathrm{TE}(\ell) : \exists \ell' \in \mathcal{L}, s \in \mathcal{S}.\ (\ell, s, \ell') \in \delta_p \wedge \mathrm{TE}(s)$,
  - $\mathrm{TR}(\ell) : \exists \ell' \in \mathcal{L}, s \in \mathcal{S}.\ (\ell', s, \ell) \in \delta_p \wedge \mathrm{TR}(s)$.

The Büchi program product and Büchi transaction are both products of a control flow automaton and a specification automaton. The key distinction lies in the asynchronous nature of defining the Büchi transaction product. One can conceptualize the products of two Büchi automatons as their concurrent executions. In the Büchi program product, this manifests as a synchronized execution, where each step taken by the control flow automaton corresponds to a step by the specification automaton, and vice versa.

In the context of Büchi transaction, the behavior is distinct. As per the transition definition $\widehat{\delta}$, the LTL automaton advances only when a *transaction step* occurs. A "transaction step" is based on the predicate $\mathrm{TR}(s)$, which indicates whether the statement s is a "transaction return" statement. The LTL automaton can only progress when $\mathrm{TR}(s)$ is satisfied, whereas the control flow automaton can advance without additional constraints. It is worth noting that while Büchi transaction resembles Büchi program product, they diverge in the definition of transition and final state. To be specific, since P4LTL is defined at the big time step, only a "transaction return" triggers the progression in the LTL automaton, and only states corresponding to a "transaction entering/return" are labeled as final states $\widehat{\mathcal{F}}$. This extension is of importance, for it enables verification at the transaction level semantics (i.e., P4LTL semantics) and substantially reduces the number of transitions and accepted traces that are semantically irrelevant to P4LTL.

Theorem 5.1. *The P4 program $\mathcal{P}$ satisfies the P4LTL specification φ if and only if the Büchi transaction $\mathcal{B} = \mathcal{P} \otimes \mathcal{A}_{\neg\varphi}$ does not have any fair and feasible trace.*

The proof of the theorem resembles the proof for Büchi program product discussed in [20]. Therefore, we refer the interested readers to the appendix for further details.

Example. To better illustrate the distinction between Büchi transaction and Büchi program product, consider the pseudocode and property shown in Figure 5. The snippet aims to decrease the packet field *hdr.cnt* by one and set the register *status* to zero if *hdr.cnt* is less than zero, or one otherwise at every packet processing. For brevity, we denote *hdr.cnt* as x and *status* as y. In the Büchi program product, each step in the control flow automaton corresponds to a step in the LTL automaton, as depicted in Figure 6. In contrast, for the Büchi transaction, edges $(l_0,q_0) \rightsquigarrow (l_1,q_1)$ and $(l_1,q_0) \rightsquigarrow (l_2,q_1)$ are removed, and assumptions not from $(l_2,*)$ to $(l_0,*)$ all become true. This is because only edges from l_2 to l_0 correspond to a step in the LTL automaton. The formula $\Diamond(x > 0) \rightarrow \Diamond(y = 0)$ does not hold under the definition of Büchi program product, as evidenced by a counterexample where x is consistently assigned a value of 1 at the loop front. That is, $x > 0$ eventually holds, but $y = 0$ never becomes true. However, the formula is satisfied under the notion of Büchi transaction, since only edges from l_2 to l_0 will trigger progression of the LTL automaton. As a result, the Büchi transaction has no fair and feasible trace, indicating the validity of the property. Intuitively, $x > 0$ is always accompanied by $y == 0$ at the end of every *transaction*, but not necessarily for every *statement*. Furthermore, the greater concision of Büchi transaction compared to the Büchi program product in Figure 6 and Figure 7 visually demonstrates that Büchi transaction reduces the number of accepted traces that are semantically irrelevant to P4LTL.

5.2 Verification Algorithm

Based on Theorem 5.1, the temporal verification of stateful P4 programs is reduced to checking whether the Büchi transaction has any fair and feasible trace. Note that the Büchi transaction is a special case of Büchi automaton. Therefore, we can leverage the existing verification algorithm for the Büchi automaton proposed in [20].

Algorithm 1 illustrates our verification framework. It takes a program $\mathcal{P}$ and a LTL formula φ as inputs, and outputs

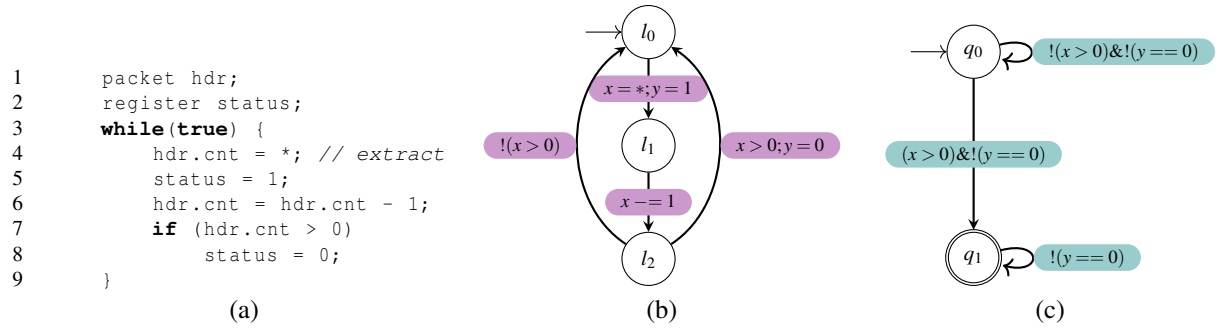

Figure 5: The program is presented in (a) as code and in (b) as control flow automaton. For brevity, we denote *hdr.cnt* as x and *status* as y. The Büchi automata representing the negation of $\Diamond(x > 0) \rightarrow \Diamond(y = 0)$ is presented in (c).

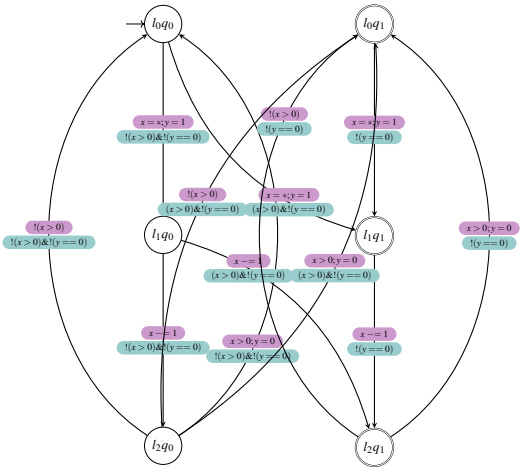

Figure 6: Büchi program product for Figure 5.

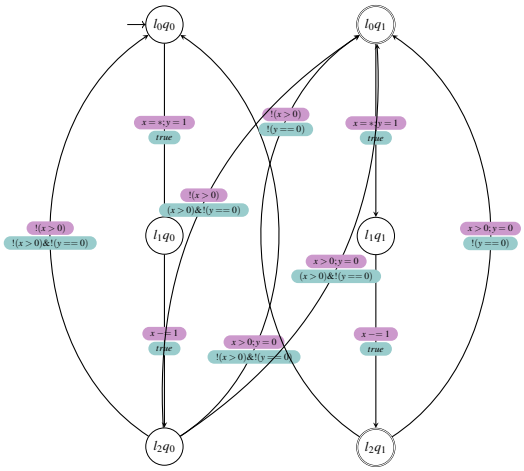

Figure 7: Büchi transaction for Figure 5.

Algorithm 1 The verification framework

Input: Program $\mathcal{P}$ and LTL property φ
Output: *Yes* if $\mathcal{P} \models \varphi$, and counterexample π otherwise

1: $\mathcal{A}_{\neg\varphi} \leftarrow$ LTL2Buchi$(\neg\varphi)$
2: $\mathcal{B} \leftarrow \mathcal{P} \otimes \mathcal{A}_{\neg\varphi}$
3: **while** $\mathcal{L}(\mathcal{B}) \neq \emptyset$ **do**
4: Let $\pi \in \mathcal{L}(\mathcal{B})$
5: **if** π is feasible **then**
6: **return** π
7: $\mathcal{B} \leftarrow$ refine$(\mathcal{B}, \pi)$
8: **return** *Yes*

the verification result. In the first place, the negation of the LTL formula $\neg\varphi$ is transformed into its corresponding Büchi automaton $\mathcal{A}_{\neg\varphi}$ (line 1). Next, the Büchi transaction is constructed to represent the counterexamples that witness the violation (line 2). The while loop (lines 4-7) aims to identify any fair and feasible trace that belongs to $\mathcal{B}$. First, it extracts a fair trace π that belongs to $\mathcal{B}$ (line 4). If π is feasible, then a counterexample is found according to Theorem 5.1. Otherwise, $\mathcal{B}$ is reduced to not accept π and other spurious traces that can be generalized from it (line 7). Finally, if there exists no fair trace for $\mathcal{B}$, the framework concludes that the property φ has been verified (line 8).

6 Implementation

The implementation of p4tv comprises approximately 10,000 lines of code in C++ and Java. As a common approach adopted by P4 verifiers [24, 40, 41, 47], we first translate P4 to an intermediate representation called Boogie [7] and proceed with the downstream verification. Next, we instrument the program with ghost variables to keep track of the terms and predicates introduced in P4LTL, similar to the approach used in p4v [40]. Our verification approach is implemented as an extension to Ultimate Automizer model checker [20]. Given a P4LTL specification, our implementation constructs the corresponding Büchi transaction introduced in Section 5 and checks for the existence of a fair and feasible trace. If the trace is discovered, the corresponding counterexample will be reported to the developer. Otherwise, p4tv will confirm that the P4LTL property is satisfied by the given P4 program.

Limitation. Currently, we note several limitations with

Name	Benchmark Description
Bfs [45]	The fast failover mechanism in a Bfs fashion to reroute packets without control plane interference.
Blink [28]	Detect characteristic failure based on information aggregated with registers and trigger rerouting.
CoDel [36]	An active queue implementation for RFC 8289.
Dfs [45]	A Dfs variation of the Bfs benchmark.
Ndn [46]	A named data networking paradigm where data are retrieved based on names rather than addresses.
P4NIS [38]	Several mechanisms for eavesdropping protection.
P4sp [37]	A sensor failure recovery system.
P4xos-acceptor [19]	P4 implementation of the acceptor in Paxos protocol.
P4xos-learner [19]	P4 implementation of the learner in Paxos protocol.

Table 1: P4 benchmark descriptions.

p4tv. Firstly, while it fully supports V1Model, it offers only limited support for TNA and PSA. Nonetheless, as demonstrated in Section 3.1, we assert that our modeling techniques remain applicable to these architectures, requiring only a more intricate design in the `P4program` process. We plan to support them in the future. Additionally, although the utilization of Büchi transactions can decrease the number of accepted traces, p4tv may still encounter the state explosion problem, which is an intrinsic challenge for model checking. Lastly, p4tv only permits developers to specify properties at the packet processing boundaries. Specifically, developers cannot directly specify or verify properties within packet processing. As per the P4LTL semantics, if a property violation occurs within packet processing but remains unobservable at the boundary, p4tv does not regard such violations as an issue.

7 Evaluation

Dataset. As p4tv pioneers temporal verification for stateful P4 programs, we introduce a dedicated dataset tailored for this purpose. This dataset comprises nine non-trivial stateful P4 programs sourced from both the open-source community and academic papers (further details provided in Table 1). For each program, we carefully analyze its design proposal and manually formulate temporal properties that we anticipate to be satisfied. Furthermore, besides several bugs that have been found in our evaluation, we intentionally inject additional artificial bugs into the control plane or code implementation to further test p4tv's bug-finding capabilities. In total, we formulate 19 stateful verification tasks.

P4 verifiers. We compare p4tv with bf4 [21], p4v [40], Vera [47], and Aquila [50], representing the state-of-the-art P4 verifiers in recent years.

All experiments were conducted on a machine with an Intel(R) Core(TM) i5-12400 2.50 GHz CPU and 32GB RAM.

7.1 Verifying Stateful Temporal Properties

Table 2 presents the evaluation results of our tool, including the specification size and detailed description of verified properties. The CPI column indicates whether the corresponding verification task requires additional control plane information. The number with an asterisk superscript denotes an artificial bug injection. Symbol ✓ represents the successful verification, and ✗ denotes a counterexample revealing a discrepancy between program behavior and the property.

Effectiveness. All of the compared verifiers (i.e., bf4 [21], p4v [40], Vera [47] and Aquila [50]) currently do not support the temporal verification of stateful P4 programs at the packet processing level. These verifiers assume the register values to be non-deterministic at the start of each packet processing, overlooking its stateful nature. This limitation restricts their ability to specify and verify temporal properties related to register state changes and history traces.

Of all the 19 stateful verification tasks, p4tv successfully verifies 14, while reporting counterexamples for the remaining. In the 9th verification task, we discover that insufficient consideration for register initialization in the P4 implementation leads to property failure. According to the P4-16 specification [16], the initial register value is non-deterministic without manual specification, potentially causing constant packet dropping. Concerning the 13th verification task, the improper implementation of the heartbeat check for the primary port results in property violation. Moreover, p4tv effectively reveals bugs through corresponding counterexamples for manual bug injections in both CPI (task 8) and codes (tasks 15 and 18). In task 8, swapping the matching action for data packet and interest packet in CPI causes a resource interest packet to reset, rather than set, the "can be forwarded" status for the corresponding data packet. P4tv identifies the violation and reports a packet processing sequence where the switch receives an interest packet followed by a data packet but fails to forward the latter. For tasks 15 and 18, p4tv helps pinpoint the injected bugs by revealing counterexamples where the port state and round registers are incorrectly updated.

Efficiency. The verification time of p4tv ranges from 10 seconds to 21 minutes, with median and average time costs of 51 seconds and 181 seconds, respectively. Memory usage ranges from 201MB to 6.1GB, with violation detection tasks tending to require more memory. Some verification tasks, such as task 8, require relatively longer verification time, attributed to numerous refinement iterations. Generally, the inherent complexity of verification tasks primarily determines the solving time. Moreover, while larger programs and specifications often lead to more complex verification tasks, it is not always the case, e.g., tasks 5 and 16.

7.2 Verifying Assertions

This experiment compares p4tv with prior tools for assertion verification. An assertion ψ is a Boolean predicate associated with a specific P4 program point, that should always evaluate to true at that point during code execution [27]. Each assertion corresponds to a particular case of P4LTL property described

Benchmark	No.	Property Description	CPI	Prog Size	Spec Size	p4tv
Bfs	1	Reroute packets are sent to parent ports only after the whole route process is complete.		355	34	✓ (51s, 1.6GB)
Blink	2	The packets are retransmitted only after the failure signal is detected over multiple flows.		745	44	✓ (154s, 1.8GB)
CoDel	3	If in normal state, the switch will not drop packets until the queuing delay exceeds the threshold.		285	40	✓ (51s, 497.9MB)
	4	If in drop state, packets will be dropped until the queuing delay resumes.			33	✓ (18s, 286.3MB)
Dfs	5	Packets are sent to parent ports only after the whole route process is complete.		306	34	✓ (47s, 201.3MB)
Ndn	6	Forwarding data packets occurs only if the switch has received corresponding interest packets before.	✓	470	50	✓ (895s, 1.4GB)
	7	After forwarding data packets, the arrival of an immediate data packet entails drop.	✓		34	✓ (274s, 1.8GB)
	8*	Bug inject: swapping matching action for data packet and interest packet in CPI. Check property 6.	✓		50	✗ (1260s, 6.1GB)
P4NIS	9	If user packets arrive infinitely often, then they will be forwarded to certain ports infinitely often.		303	82	✗ (185s, 2.7GB)
	10	Egress ports for packets from the same source are not fixed.			40	✓ (39s, 1.8GB)
P4sp	11	When timeout is detected, the switch will forward packets from the redundant port until the primary port recovers.		222	64	✓ (14s, 623.8MB)
	12	If a packet from the primary port is forwarded, packets below the threshold will be dropped next time.			38	✓ (22s, 836.0MB)
	13	Early packets from the secondary port are forwarded only if the time threshold has been exceeded before.			38	✗ (64s, 1.7GB)
	14	If a packet from the primary port arrives, the forwarding decision will depend on the corresponding timestamp until the arrival of the next primary port packet.			39	✓ (10s, 513.1MB)
	15*	Bug inject: The state update action is interchanged between primary pory and secondary port. Check property 12.		231	38	✗ (85s, 1.9GB)
Paxos-acceptor	16	After a p4xos packet is forwarded, the p4xos packets of earlier rounds will not be forwarded by the switch.	✓	355	66	✓ (45s, 857.9MB)
	17	The acceptor switch only votes for the value it stores until a new election packet arrives.			31	✓ (28s, 719.1MB)
	18*	Bug inject: Wrongly updating the value round instead of the current valid round for round register. Check property 17.			31	✗ (176s, 2.8GB)
Paxos-learner	19	The learner switch will not forward votes until the vote has reached majority agreements.		322	43	✓ (21s, 396.5MB)

Table 2: Results of stateful temporal verification.

by $\square\psi$, indicating that "the predicate ψ should hold at the end of every packet processing." We can use p4tv to verify assertion by instrumenting a shadow variable to track the assertion's value at that point and initializing it with *true* at the beginning of each packet processing stage.

Given Vera's limited support for user-defined assertions and Aquila's not publicly accessible, we contrast p4tv with bf4 and p4v, the current state-of-the-art P4 verifiers. We utilize the publicly available code for bf4 and employ the approximate implementation for p4v in [21], as p4v is not open-sourced. Furthermore, it's important to note that while assertions are supported by prior tools, they operate under a model that assumes all register values to be nondeterministic and only considers a single packet processing. In the subsequent discussion, an assertion is deemed *stateful* (or *stateless*) based on whether it is related to registers or not.

Table 3 presents the results of the assertion verification. The symbol ✓ indicates successful validation of the property, ✗ signifies a genuine counterexample, and Δ denotes that the tool returns a spurious counterexample, which would not occur under the given configuration. SimpleNat is a simple stateless NAT P4 implementation. For conciseness, the results of all six invalid header access checks are combined into one line since their verification tasks are nearly identical as well as their verification time. The results of valid header access checks are similar.

As depicted in Table 3, p4tv is able to verify stateless assertions as the priors do. It's worth noting that while its performance seems relatively degraded, this is attributable to the fact that p4tv evaluates assertions within the multiple-packet model, introducing additional complexity to the verification process. Regarding stateful assertions, p4tv effectively con-

Benchmark	Assertion	Stateful?	p4tv	bf4	p4v
SimpleNat-bug1_6	Whether the header is valid before accessing.	No	✗ (2m)	✗ (3s)	✗ (4s)
SimpleNat-safe1_11			✓ (10s)	✓ (3s)	✓ (4s)
P4NIS-safe1	The egress port should be equal to the register value if tunnel is applied.	Yes	✓ (2m36s)	✓ (3s)	✓ (2s)
P4NIS-safe2	Labeled packets cannot be implicitly dropped.	Yes	✓ (59s)	Δ(3s)	Δ(3s)
P4xos-acceptor-safe	Stored consensus round must be always less than the current valid round.	Yes	✓ (28m21s)	Δ(4s)	Δ(3s)
P4xos-learner-unsafe	The value of vote counter register is always zero.	Yes	✗ (6m54s)	Δ(2s)	Δ(3s)
P4xos-acceptor-unsafe	Stored consensus round must be always equal to the current valid round.	Yes	✗ (3m40s)	Δ(3s)	Δ(2s)

Table 3: Results of assertion verification.

firms safe assertions and reports genuine counterexamples for unsafe ones. Conversely, bf4 and p4v, assuming registers to be entirely non-deterministic, exhibit incompleteness in verifying assertions under specific register configurations. For instance, under the zeroing register configuration, the assertion $vround[hdr.inst] \leq round[hdr.inst]$ for P4xos-acceptor-safe remains valid throughout subsequent processing, but may easily fail under the non-deterministic configuration. Furthermore, for benchmarks like P4xos-learner-unsafe, although bf4 and p4v successfully invalidate the assertion, they reported counterexamples that would never occur under the given configuration. In conclusion, the overapproximation of the priors results in imprecision in verifying stateful properties.

7.3 Scalability Evaluation

In this section, we explore the scalability of p4tv in two dimensions: program complexity and specification complexity.

To assess the scalability of p4tv concerning program complexity, we introduce a parameter named PORT_NUM for P4NIS to parameterize the range of valid ports. We verify the following valid P4LTL property: if the register *count* is set within PORT_NUM, it will remain within this range for all subsequent processing. Since P4 is a loop-free language, increasing PORT_NUM by 1 entails adding an additional if block to the program that updates the *count* register and sets the metadata, without expanding the size of the P4LTL property. The verification results are presented in Figure 8.

To examine the scalability of p4tv concerning specification complexity, we conduct experiments on P4xos-acceptor. An essential property of the acceptor is that the round recording the consensus value must always be less than or equal to the current valid round, as expressed by the P4LTL property:

$$\wedge_{i=0}^{\mathrm{CNT}} vround[i] \leq round[i] \rightarrow \Box(vround[i] \leq round[i]).$$

This ensures that outdated messages are disregarded, which is crucial for achieving consensus. We verify P4xos-acceptor with the property as the parameter CNT increases. The verification results are illustrated in Figure 9.

As indicated in Figure 8 and Figure 9, with the other dimension fixed, the verification time of p4tv exhibits rapid growth as the complexity factors PORT_NUM and CNT become relatively large. Moreover, the validity of the property also impacts the verification time. For instance, manually substituting $\leq$ with $>$ in every clause of the P4xos-acceptor specification, the time required to return a counterexample remains nearly identical for CNT $= 8$ and CNT $= 16$ (in about 90s), since constructing a counterexample trace is straightforward. Ultimately, the root cause of verification time is the intrinsic complexity of the verification task.

8 Case Studies

P4tv can verify a wide range of temporal properties within stateful P4 programs. In this section, we present our verifying experiences with p4tv through some case studies.

P4NIS [38]. Processors-based network immune scheme (P4NIS) aims to protect the network against packet eavesdropping within the programmable data plane. The snippet of P4NIS is presented previously in Section 2. An important liveness property associated with the defense mechanism can be expressed as follows:

If specific packets keep arriving and being forwarded, they should be forwarded to ports 1, 2 and 3 infinitely often.

We can use P4LTL to concisely express the temporal property. Firstly, it is necessary to define the term "specific packets" within the context of P4NIS. These packets are identified as those with an IPv4 destination address other than the domain name server (123.123.123.123) and received from port 0. Moreover, the environment assumption in P4LTL can be leveraged to describe the "If" part of the property. Specifically, the environment assumption can be formulated as $\varphi_E = \Box\Diamond\phi_{sp}$, where

$$\phi_{sp} = standard\_metadata.ingress\_port == 0 \\ \wedge \mathrm{old}(hdr.ipv4.dstAddr) \neq 123.123.123.123,$$

denoting the "specific" condition in the context of P4NIS mechanism. Moreover, we utilize ϕ_{sp} to specify the subsequent portion of the property, which states that "they should be forwarded to ports 1, 2 and 3 infinitely often" as follows:

$$\varphi_P = \bigwedge_{i=1}^{3} \Box\Diamond(\phi_{sp} \rightarrow \mathrm{fwd}(i))$$

In summary, we can use the formula $\varphi = \varphi_E \rightarrow \varphi_P$ to formalize the temporal property of such stateful functionalities. However, the property φ does not hold. Upon inputting φ

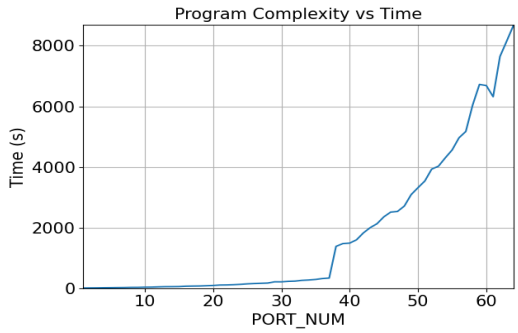

Figure 8: Verification time curve as PORT_NUM grows.

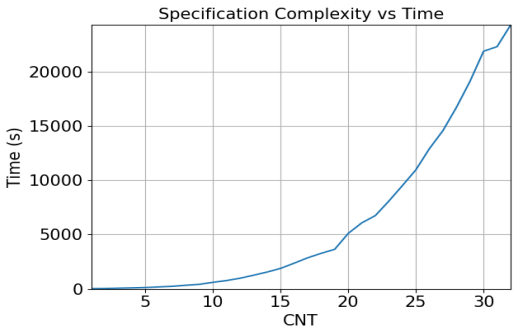

Figure 9: Verification time curve as CNT grows.

and the P4NIS code, p4tv will report a counterexample indicating that the value of the port record register `count[0]` is improperly configured. The oversight of register configuration leads to the potential of a packet skipping all the conditional branches and being implicitly dropped (i.e., the forwarding port *egress_spec* is not set). The intrinsic issue is that the P4 language specification [16] assumes that the default value of the register `count[0]` is undefined. Consequently, a switch deployed with the P4NIS mechanism may drop all the receiving packets when `count[0]` is configured with a value outside the range $[0,2]$, e.g., configured with 4.

To fix the potential vulnerability, we can adopt two strategies regarding different aspects. Suppose we are not certain about whether the underlying hardware performs the default zero configuration. In that case, we can amend the code with an additional branch condition, resetting `count[0]` when it is improperly configured.

```
1       count.read(temp,(bit<32>)0);
2       // set count back to normal value
3       if(temp > 2){
4           count.write((bit<32>)0, (bit<32>)1);
5           standard_metadata.egress_spec = 1;
6       }
```

If we have a certain confidence in the hardware configuration, the property part φ_P becomes

$$\square(count[0] \geq 0 \wedge count[0] \leq 2 \rightarrow \bigwedge_{i=1}^{3} \square\Diamond(\phi_{sp} \rightarrow \text{fwd}(i))),$$

which states that when `count` is appropriately configured, packets will be forwarded to ports 1,2 and 3 in a circular way. Either application of the strategies will enable p4tv to verify the correctness of the specification, proving that the temporal property is satisfied by P4NIS.

NDN [46]. Named-Data Networking (NDN) implements a content-centric networking paradigm that aims to provide an efficient and scalable way to manage distributed storage, as opposed to the typical destination-centric approach.

NDN employs two packet types, interest and data, to implement its content-centric paradigm. Interest packets represent requests, while data packets represent responses. If the request name in an interest packet is not matched in the Forwarding Information Base (FIB) table, the switch will drop the packet. Conversely, if the request name is present in the FIB, the switch will store the incoming port information in the Pending Interest Table (PIT) and forward the packet based on the port information stored in the FIB.

If a data packet experiences a miss in the Pending Interest Table (PIT), the switch will drop it. This occurs because the corresponding pending request is not logged. Conversely, if the data packet matches an entry in the PIT, the switch will forward the packet through the port where the request packet originates from and remove the entry. Both the Forwarding Information Base (FIB) and Pending Interest Table (PIT) are using registers as keys.

Based on the NDN implementation, its property φ_{ndn} can be formulated in P4LTL as

$$\square(\phi_{data} \wedge \neg\text{drop} \rightarrow \circ((\phi_{data} \rightarrow \text{drop})\,\mathcal{W}(\phi_{int} \wedge \neg\text{drop}))),$$

where $meta.nameHash == a \wedge meta.packetType == DATA$ is represented by ϕ_{data}, and ϕ_{int} denotes $meta.nameHash == a \wedge meta.packetType == INT$.

The specification φ_{ndn} states that, for any given name hash a, the switch will always drop the data packet until it forwards the corresponding request packet. Note that a serves as a free variable, indicating that the specification is expected to hold for any name hash. Additionally, we use $\phi_{data} \rightarrow$ drop to specify that we only expect the specification to hold after a data packet has been forwarded, since the corresponding PIT entry is cleared afterwards.

Upon inputting the specification and the NDN code, p4tv returns a counterexample. The reason for this outcome is that the table entries control a part of the key forwarding logic. The property does not hold if we assume that the table entries are arbitrary. To address the problem, we employ CPI to complement the forwarding logic, stating that if the requesting data is not recorded in PIT or FIB, the tables `routeData_table` and `updatePit_table` should apply the action drop, and otherwise the table `updatePit_table` should update the PIT table accordingly, as shown below:

$$\begin{aligned}&\square(\text{apply}(ut) \wedge \text{key}(ut, meta.FIB) == 0 \rightarrow \text{apply}(ut, drop)\\ &\wedge \text{apply}(ut) \wedge \text{key}(ut, meta.FIB) \neq 0 \rightarrow \text{apply}(ut, update)\\ &\wedge \text{apply}(rt) \wedge \text{key}(rt, meta.PIT) == 0 \rightarrow \text{apply}(rt, drop))\end{aligned}$$

After inputting the CPI along with the NDN implementation, p4tv verifies the correctness of the property.

9 Related Work

SDN verification. The verification for SDN can be classified into control plane and data plane verification.

Traditionally, studies analyzing the control plane have focused on topological properties such as reachability and isolation. Arc [26] facilitates rapid control plane verification without the data plane generation. Batfish [23] introduces a general approach to detect errors in network configurations by deriving the data plane effectively. CrystalNet [39] offers a proactive means of validating network operations using emulation techniques. Minesweeper [8] converts network configurations into logical formulas and checks for satisfiability to identify any network state that violates the specified query. These studies primarily analyze network configurations and strive to address queries related to network-wide properties.

Data plane verification tools address similar queries by requiring a snapshot of the network's data plane. Numerous techniques have been proposed to verify the data plane from various perspectives. Anteater [42] employs graph theory algorithms to verify network-wide properties, representing the network as a graph. HSA [33] develops an optimized formalism to statically check network configurations with specifications and identify network failures. Veriflow [34] aims to enable real-time validation of network-wide invariants, while Symnet [48] validates network-wide queries by injecting symbolic packets and tracking their behavior through the network. Our tool, p4tv, falls within the category of data plane verification tools and is specifically designed for verifying processing-level temporal properties of stateful P4 programs.

P4 verification tools. Aquila [50], bf4 [21] and p4v [40] encode P4 programs with SMT formulas and verify the properties using constraint solving. ASSERT-P4 [24] and Vera [47] use symbolic execution to verify P4 programs. However, these P4 verifiers overlook the stateful nature of packet processing. They assume register states are nondeterministic before each packet processing, which fundamentally hinders them from verifying the behaviors of stateful P4 programs. Moreover, Aquila, ASSERT-P4, bf4, and bf4 use Hoare logic and Vera proposes NetCTL that is a subset of temporal logic language CTL [13]. Both Hoare logic and NetCTL focus on properties of an individual packet processing and do not consider historical information. On the contrary, p4tv considers the preservation of register states in its execution model, and provides P4LTL to support processing-level temporal specification.

Stateful network verification tools. Several studies have previously addressed the verification of stateful networks, but they have focused on different aspects compared to p4tv. For instance, [54] concentrates on the behaviors of match-action tables in SDN, while [5] verifies congestion control protocols by transforming network models into SMT formulas. Previous works such as [1, 2, 52] utilize abstract interpretation to verify stateful networks, with a focus on network-level properties such as isolation. These works fall under the category of control plane verification. In contrast, p4tv targets data plane verification with its focus on P4. Aragog [53] aims to verify distributed network functions, but is limited to verifying network invariants and run-time checks. P4wn [31] is a symbolic execution tool for stateful P4 programs that reports the probability of each program path. In our work, we adopt a similar approach as P4wn, treating stateful P4 programs as infinite loops. However, we enhance this approach by incorporating additional register initializations to adapt it to our specific verification scenario. Notably, our research question differs from P4wn as our primary objective is *formal verification* rather than *statistical testing* for P4 programs.

Temporal verification. There have been a number of approaches to verifying the temporal properties of state-based systems [6, 12, 14, 15, 17, 18, 20, 29]. Several approaches [14, 15, 17, 18, 20] employ counterexample-guided abstraction refinement to verify temporal properties. They refine the abstract model iteratively utilizing spurious counterexamples encountered until no counterexample exists or a real counterexample is discovered. In our verification approach, we adopt the framework proposed by [20]. However, it is important to note that our approach differs in several aspects. Firstly, we aim at different semantics for temporal properties. Instead of verifying properties at the small time step, we focus on checking them at the big time step, which limits the direct application of the common approach to P4LTL. Moreover, for verification purposes, we construct Büchi transaction rather than Büchi program product. This imposes additional constraints on the transitions and final states, resulting in a reduced number of transitions and accepted traces and making the verification more practical.

10 Conclusions

Current P4 verifiers simplify P4 verification by overapproximating register states, which leads to challenges in specifying and verifying complex stateful P4 programs. We present p4tv, the first automated tool for temporal verification of stateful P4 programs at the packet processing level. We introduce a specification language called P4LTL to describe stateful functionalities and formulated Büchi transaction to make the verification more tractable. The evaluation of verification tasks demonstrates p4tv's effectiveness in verifying temporal properties of stateful P4 programs. P4tv is available at https://thufv.github.io/research/p4tv.

Acknowledgment

This work was supported in part by the National Natural Science Foundation of China (No. 62072267 and No. 62021002).

References

[1] Kalev Alpernas, Roman Manevich, Aurojit Panda, Mooly Sagiv, Scott Shenker, Sharon Shoham, and Yaron Velner. Modular safety verification for stateful networks. *CoRR*, abs/1708.05904, 2017.

[2] Kalev Alpernas, Roman Manevich, Aurojit Panda, Mooly Sagiv, Scott Shenker, Sharon Shoham, and Yaron Velner. Abstract interpretation of stateful networks. In Andreas Podelski, editor, *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, volume 11002 of *Lecture Notes in Computer Science*, pages 86–106. Springer, 2018.

[3] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: semantic foundations for networks. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 113–126. ACM, 2014.

[4] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. SNAP: stateful network-wide abstractions for packet processing. In Marinho P. Barcellos, Jon Crowcroft, Amin Vahdat, and Sachin Katti, editors, *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 29–43. ACM, 2016.

[5] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. Toward formally verifying congestion control behavior. In Fernando A. Kuipers and Matthew C. Caesar, editors, *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021*, pages 1–16. ACM, 2021.

[6] Jiri Barnat, Lubos Brim, Vojtech Havel, Jan Havlícek, Jan Kriho, Milan Lenco, Petr Rockai, Vladimír Still, and Jirí Weiser. Divine 3.0 - an explicit-state model checker for multithreaded C & C++ programs. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 863–868. Springer, 2013.

[7] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[8] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 155–168. ACM, 2017.

[9] Cristian Hernandez Benet, Andreas J. Kassler, Theophilus Benson, and Gergely Pongrácz. MP-HULA: multipath transport aware load balancing using programmable data planes. In Xin Jin and Changhoon Kim, editors, *Proceedings of the 2018 Morning Workshop on In-Network Computing, NetCompute@SIGCOMM 2018, Budapest, Hungary, August 20, 2018*, pages 7–13. ACM, 2018.

[10] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. *Comput. Commun. Rev.*, 44(3):87–95, 2014.

[11] Marco Canini, Daniele Venzano, Peter Peresíni, Dejan Kostic, and Jennifer Rexford. A NICE way to test openflow applications. In Steven D. Gribble and Dina Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 127–140. USENIX Association, 2012.

[12] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.

[13] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[14] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July*

15-19, 2000, Proceedings, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[15] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[16] P4 Langhuage Consortium. $P4_{16}$ language specifcation, 2023.

[17] Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. Proving that programs eventually do something good. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 265–276. ACM, 2007.

[18] Byron Cook and Eric Koskinen. Making prophecies with decision predicates. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 399–410. ACM, 2011.

[19] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a network service. *IEEE/ACM Trans. Netw.*, 28(4):1726–1738, 2020.

[20] Daniel Dietsch, Matthias Heizmann, Vincent Langenfeld, and Andreas Podelski. Fairness modulo theory: A new approach to LTL software model checking. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 49–66. Springer, 2015.

[21] Dragos Dumitrescu, Radu Stoenescu, Lorina Negreanu, and Costin Raiciu. bf4: towards bug-free P4 programs. In Henning Schulzrinne and Vishal Misra, editors, *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, pages 571–585. ACM, 2020.

[22] Nick Feamster, Jennifer Rexford, and Ellen W. Zegura. The road to SDN: an intellectual history of programmable networks. *Comput. Commun. Rev.*, 44(2):87–98, 2014.

[23] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd D. Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, pages 469–483. USENIX Association, 2015.

[24] Lucas Freire, Miguel C. Neves, Lucas Leal, Kirill Levchenko, Alberto E. Schaeffer Filho, and Marinho P. Barcellos. Uncovering bugs in P4 programs with assertion-based verification. In *Proceedings of the Symposium on SDN Research, SOSR 2018, Los Angeles, CA, USA, March 28-29, 2018*, pages 4:1–4:7. ACM, 2018.

[25] Paul Gastin and Denis Oddoux. Fast LTL to büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2001.

[26] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In Marinho P. Barcellos, Jon Crowcroft, Amin Vahdat, and Sachin Katti, editors, *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 300–313. ACM, 2016.

[27] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[28] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. Blink: Fast connectivity recovery entirely in the data plane. In Jay R. Lorch and Minlan Yu, editors, *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 161–176. USENIX Association, 2019.

[29] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[30] Enio Kaljic, Almir Maric, Pamela Njemcevic, and Mesud Hadzialic. A survey on data plane flexibility and programmability in software-defined networking. *IEEE Access*, 7:47804–47840, 2019.

[31] Qiao Kang, Jiarong Xing, Yiming Qiu, and Ang Chen. Probabilistic profiling of stateful data planes for adversarial testing. In Tim Sherwood, Emery D. Berger, and Christos Kozyrakis, editors, *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 286–301. ACM, 2021.

[32] Naga Praveen Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: scalable load balancing using programmable data planes. In Brighten Godfrey and Martín Casado, editors, *Proceedings of the Symposium on SDN Research, SOSR 2016, Santa Clara, CA, USA, March 14 - 15, 2016*, page 10. ACM, 2016.

[33] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In Steven D. Gribble and Dina Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 113–126. USENIX Association, 2012.

[34] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and Philip Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 15–27. USENIX Association, 2013.

[35] Ekkart Kindler. Safety and liveness properties: A survey. *Bulletin of the European Association for Theoretical Computer Science*, 53(268-272):30, 1994.

[36] Ralf Kundel, Jeremias Blendin, Tobias Viernickel, Boris Koldehofe, and Ralf Steinmetz. P4-codel: Active queue management in programmable data planes. In *IEEE Conference on Network Function Virtualization and Software Defined Networks, NFV-SDN 2018, Verona, Italy, November 27-29, 2018*, pages 1–4. IEEE, 2018.

[37] Steffen Lindner, Marco Häberle, Florian Heimgaertner, Naresh Nayak, Sebastian Schildt, Dennis Grewe, Hans Löhr, and Michael Menth. P4 in-network source protection for sensor failover. In *2020 IFIP Networking Conference, Networking 2020, Paris, France, June 22-26, 2020*, pages 791–796. IEEE, 2020.

[38] Gang Liu, Wei Quan, Nan Cheng, Ning Lu, Hongke Zhang, and Xuemin Shen. P4NIS: improving network immunity against eavesdropping with programmable data planes. In *39th IEEE Conference on Computer Communications, INFOCOM Workshops 2020, Toronto, ON, Canada, July 6-9, 2020*, pages 91–96. IEEE, 2020.

[39] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 599–613. ACM, 2017.

[40] Jed Liu, William T. Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Calin Cascaval, Nick McKeown, and Nate Foster. p4v: practical verification for programmable data planes. In Sergey Gorinsky and János Tapolcai, editors, *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, pages 490–503. ACM, 2018.

[41] Nuno Lopes, Nikolaj Bjørner, Nick McKeown, Andrey Rybalchenko, Dan Talayco, and George Varghese. Automatically verifying reachability and well-formedness in p4 networks. *Technical Report, Tech. Rep*, 2016.

[42] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In Srinivasan Keshav, Jörg Liebeherr, John W. Byers, and Jeffrey C. Mogul, editors, *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Toronto, ON, Canada, August 15-19, 2011*, pages 290–301. ACM, 2011.

[43] Rui Miguel, Salvatore Signorello, and Fernando M. V. Ramos. Named data networking with programmable switches. In *2018 IEEE 26th International Conference on Network Protocols, ICNP 2018, Cambridge, UK, September 25-27, 2018*, pages 400–405. IEEE Computer Society, 2018.

[44] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[45] Roshan Sedar, Michael Borokhovich, Marco Chiesa, Gianni Antichi, and Stefan Schmid. Supporting emerging applications with low-latency failover in P4. In Dipankar Raychaudhuri and Richard Li, editors, *Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies, NEAT@SIGCOMM 2018, Budapest, Hungary, August 20, 2018*, pages 52–57. ACM, 2018.

[46] Salvatore Signorello, Radu State, Jérôme François, and Olivier Festor. Ndn.p4: Programming information-centric data-planes. In *IEEE NetSoft Conference and Workshops, NetSoft 2016, Seoul, South Korea, June 6-10, 2016*, pages 384–389. IEEE, 2016.

[47] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging P4 programs with vera. In Sergey Gorinsky and János Tapolcai, editors, *Proceedings of the 2018 Conference of the ACM*

Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018, pages 518–532. ACM, 2018.

[48] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Symnet: Scalable symbolic execution for modern networks. In Marinho P. Barcellos, Jon Crowcroft, Amin Vahdat, and Sachin Katti, editors, *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 314–327. ACM, 2016.

[49] Kausik Subramanian, Anubhavnidhi Abhashkumar, Loris D'Antoni, and Aditya Akella. D2R: dataplane-only policy-compliant routing under failures. *CoRR*, abs/1912.02402, 2019.

[50] Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, Jie Lu, Xionglie Wei, Hongqiang Harry Liu, Ming Zhang, Chen Tian, and Minlan Yu. Aquila: a practically usable verification system for production-scale programmable data planes. In Fernando A. Kuipers and Matthew C. Caesar, editors, *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021*, pages 17–32. ACM, 2021.

[51] Moshe Y. Vardi. Branching vs. linear time: Final showdown. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, page 1–22, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[52] Yaron Velner, Kalev Alpernas, Aurojit Panda, Alexander Rabinovich, Mooly Sagiv, Scott Shenker, and Sharon Shoham. Some complexity results for stateful network verification. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*, pages 811–830. Springer, 2016.

[53] Nofel Yaseen, Behnaz Arzani, Ryan Beckett, Selim Ciraci, and Vincent Liu. Aragog: Scalable runtime verification of shardable networked systems. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 701–718. USENIX Association, 2020.

[54] Farnaz Yousefi, Anubhavnidhi Abhashkumar, Kausik Subramanian, Kartik Hans, Soudeh Ghorbani, and Aditya Akella. Liveness verification of stateful network functions. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 257–272. USENIX Association, 2020.

[55] Xiaoquan Zhang, Lin Cui, Kaimin Wei, Fung Po Tso, Yangyang Ji, and Weijia Jia. A survey on stateful data plane in software defined networks. *Comput. Networks*, 184:107597, 2021.

Appendix

A Theorem Proof

Theorem A.1. *The P4 program $\mathcal{P}$ satisfies the P4LTL specification ϕ if and only if the Büchi transaction $\mathcal{B} = \mathcal{P} \otimes \mathcal{A}_{\neg\varphi}$ does not have any fair and feasible trace.*

Proof. To begin with, we introduce the general idea of the theorem at the high level. The Büchi transaction $\mathcal{B} = \mathcal{P} \otimes \mathcal{A}_{\neg\varphi}$ corresponds to the automaton that represents the set of every *counterexample*. In other words, every fair and feasible trace of $\mathcal{B}$ corresponds to a counterexample indicating that $\mathcal{P} \not\models \varphi$. Next, we present the sketch of the theorem proof.

For every trace $\pi = (s_0, \text{assume } a_0)\ (s_1, \text{assume } a_1)\ \cdots$ that is fair and feasible to $\mathcal{B}$, we can extract the statement part $\pi_{\mathcal{P}} = s_0\ s_1 \cdots$ and the LTL part $\pi_{\mathcal{A}} = a_0\ a_1 \cdots$. For the LTL part, we can reduce $\pi_{\mathcal{A}}$ to $\pi'_{\mathcal{A}}$ by deleting the *true* characters imposed by Büchi transaction, and use the definition of Büchi transaction to show that π is fair and feasible to $\mathcal{B}$ if and only if $\pi_{\mathcal{P}}$ is fair and feasible to $\mathcal{P}$, $\pi'_{\mathcal{A}}$ belongs to $\mathcal{A}_{\neg\varphi}$ and $\pi_{\mathcal{P}} \models \pi'_{\mathcal{A}}$.

Moreover, we prove that P4 program $\mathcal{P}$ satisfies the P4LTL specification ϕ if the Büchi transaction $\mathcal{B} = \mathcal{P} \otimes \mathcal{A}_{\neg\varphi}$ does not have any fair and feasible trace. We prove it by contradiction. If it were false, then $\mathcal{P} \models \varphi$ and there exists an fair and feasible trace of $\mathcal{B}$. Therefore, we know that there also exists $\pi_{\mathcal{P}}$ and $\pi_{\mathcal{A}}$ such that $\pi_{\mathcal{P}}$ is fair and feasible to $\mathcal{P}$, $\pi'_{\mathcal{A}}$ belongs to $\mathcal{A}_{\neg\varphi}$ and $\pi_{\mathcal{P}} \models \pi'_{\mathcal{A}}$. According to the semantics of P4LTL, $\pi_{\mathcal{P}}$ can serve as a witness to the violation of φ since the execution conform to the formula $\neg\varphi$, and therefore leads to a contradiction.

Furthermore, we prove that P4 program $\mathcal{P}$ satisfies the P4LTL specification ϕ only if the Büchi transaction $\mathcal{B} = \mathcal{P} \otimes \mathcal{A}_{\neg\varphi}$ does not have any fair and feasible trace. Assume that $\mathcal{B}$ does not have any fair and feasible trace while $\mathcal{P}$ does not satisfy the P4LTL specification ϕ. According to the semantics of P4LTL, there must exists a fair and feasible trace $\pi_{\mathcal{P}} = s_0\ s_1 \cdots \in \mathcal{P}$ such that $\pi_{\mathcal{P}} \models \neg\varphi$. Moreover, there also exists a corresponding $\pi'_{\mathcal{A}} = a_0\ a_1 \cdots \in \mathcal{A}_{\neg\varphi}$ that witnesses the violation. We then reduce $\pi_{\mathcal{P}}$ to $\pi'_{\mathcal{P}} = s'_0\ s'_1 \cdots$ by deleting statements that do not satisfy $\text{TE}(s) \vee \text{TR}(s)$, and aligns

$\pi'_{\mathcal{P}}$ with $\pi'_{\mathcal{A}}$ to form $\pi' = (s'_0, \text{assume } a'_0)\ (s_1, \text{assume } a'_1)\ \cdots$. Next, we construct $\pi = (s_0, \text{assume } a_0)\ (s_1, \text{assume } a_1)\ \cdots$ by recovering $(s, \text{assume } \mathit{true})$ for every deleted statement s from $\pi_{\mathcal{P}}$. The trace π is fair and feasible to $\mathcal{B}$ based on the definition of Büchi transaction, and therefore contradicts the assumption. □